An Exception Handling Framework for N-Version Programming in Object Oriented Systems

A. Romanovsky

TECHNICAL REPORT SERIES

No. CS-TR-684
November 1999

Contact:
alexander.romanovsky@ncl.ac.uk
www.cs.ncl.ac.uk/people/alexander.romanovsky/home.formal

An Exception Handling Framework for N-Version Programming in Object Oriented Systems

Alexander Romanovsky
Department of Computing Science
University of Newcastle upon Tyne

Structuring complex systems out of components with both normal and exceptional outcomes and using exception handling for dealing with abnormal events are well-accepted practices. This paper proposes an approach for introducing exception handling into object-oriented N-version programming (NVP). We start with outlining general principles of structuring systems with diversity and show why it is important to use exceptions while developing and using diversely-developed software. Internal version exceptions and external exceptions, which the diversely-designed class can propagate, are clearly separated in our framework: each version has its own internal exceptions but the external exceptions of all versions have to be the same and identical to the interface exceptions of the whole class. This scheme requires an adjudicator of a special kind to allow interface exception signalling when a majority of versions have signalled the same exception. We demonstrate these ideas using a general framework for introducing NVP into object-oriented systems which we have developed recently [1]. This framework follows all principles of structured NVP: software diversity is introduced here at the level of classes and encapsulated into the diversely-designed class. We discuss the internal structure of this class and the interfaces of its subcomponents; and show how the NVP controller works, version execution is coordinated and re-use operates here. This framework makes use of many advantages object-oriented programming has. For the demonstration, it has been implemented in Ada. The paper finishes with a comparison of our proposal with some existing NVP schemes and with a discussion of our future work.

1. Introduction

1.1. Structuring Complex Systems. Components, Interfaces, Exceptions

Complex systems are always designed of components which encapsulate some parts of system data and behaviour and are accessed via interfaces. Exceptions (abnormal situations) and exception handling are parts of many models used at different phases of the life cycle. Exception handling is a structuring mechanism which allows us to clearly separate normal and abnormal behaviour of the system [2] [3]. It is used in designing many modern industrial applications. Many object-oriented languages include exception handling features: Ada [4], Java, C++, etc. For example, one can design a class list in Ada using a detailed set of exceptions:

```
package list_class is
   type list_t is abstract tagged limited null record;
   -- application methods:
```

```
   procedure Initialize (l : access list_t) is abstract;
   procedure Insert (l : access list_t; elem : in elem_t) is abstract;
   procedure Remove (l : access list_t; elem : in elem_t) is abstract;
   procedure Get_Min (l : access list_t; elem : out elem_t) is abstract;
   procedure Sort (l : access list_t) is abstract;
   function Check (l : access list_t; elem : in elem_t) return boolean is abstract;
   -- interface exceptions:
   list_empty : exception;          -- Remove; Get_Min; Check; Sort
   list_full : exception;           -- Insert
   list_absent_element : exception; -- Remove
   list_failure : exception; -- Initialize; Insert; Remove; Get_Min; Sort; Check
end list_class ;
```

Exception handling offers several ways for separating normal and abnormal behaviour: it separates not only the code (handlers from the normal code), but the normal control flow from the exceptional one. The latter includes a separate execution of the normal code and of exception handlers, and two ways of returning the control flow to a component after it has used another component. This is why there are interface exceptions in any exception handling scheme. In practice components can have a very sophisticated set of interface exceptions to be propagated outside. Any component using other components has to be aware of these exceptions and to be ready to handle them should they be propagated.

This is a common practice in developing components of different sorts, including libraries, classes, modules. For example, in Ada the main structuring blocks are packages (classes are built as their extensions) which have specifications and implementations. Package specifications include interface procedures/functions and exceptions. The body of each subprogram may have several internal exceptions declared, each of which can have a handler designed as part of the subprogram. If there is no handler for a particular exception or the subprogram fails to handle it, an interface exception is propagated outside the package.

To define the rules of exception handling and, in particular, exception propagation, one has to associate exception handling with a *structuring technique*. A set of exceptions and exception handlers is associated with an *exception context* (which can be, for example, a procedure, a block, a class, or a package). If one cannot handle an exception raised within the context or if there is no handler for the exception raised, then an exception (a different one or the same) is propagated to the containing context where a corresponding handler will be called. Each context is associated with a component. We will say that, for a given component, the *containing context* is associated with the component which uses it (or that the former component is associated with the *nested context* with respect to the latter). "Uses" means here that the component refers to the interface of another component. We can consider dynamic system structuring based on nested procedure/method calls as an example: many procedural or module languages use the stack of nested calls for exception propagation (Ada, CLU [5]).

1.2. N-Version Programming

Software diversity has proved to be the most general way of coping with residual software faults. Two approaches were proposed in the 70-ies and have been investigated thoroughly since then. They are recovery blocks [6] and N-version programming [7].

We do not intend to compare them here (for that, readers are referred to [8]), but we would like to note that each of these approaches has its own pros and cons, has been used in practice several times, and that, generally speaking, they are the only general ways to secure tolerating software faults. In this paper we will concentrate on N-version programming (NVP). In this approach, the N *versions* of a program (a module) are developed independently by different programmers to be run concurrently. Their results are compared by an *adjudicator*. The simplest way is to use the majority voting here: the results produced by the majority of versions are assumed to be correct, the rest of the versions are assumed to have errors, their faults having been triggered in the execution. This technique requires a special support to control the execution of versions and the adjudicator and pass the information among them (we refer to it as the *controller*). In particular, it synchronises the version execution to obtain information (e.g. results) from all of them to pass to the adjudicator. Figure 1 shows how these components work together. The functionalities of the controller in the first experiments with NVP were executed by and hidden in a special run-time support DEDIX [7].

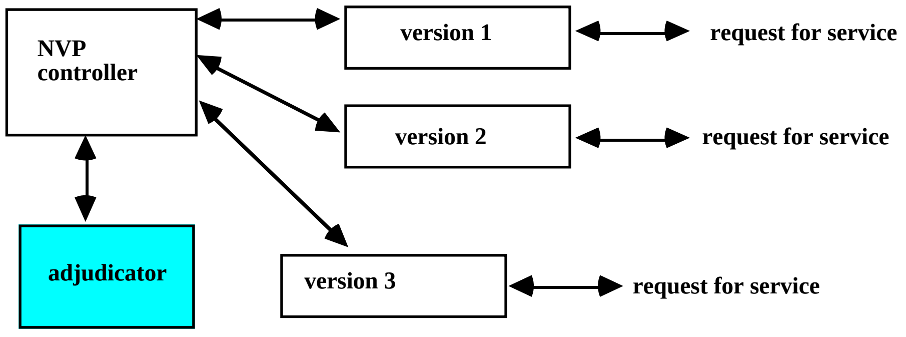

Figure 1. Subcomponents of NVP

Independent version design is vital in applying the NVP scheme because version designers tend to make similar mistakes which can cause several versions to fail on the same inputs. Special methodologies to help ensure this independence are proposed in [9].

1.3. Structuring Systems with Diversity

Many researchers realise that using software diversity schemes can be dangerously error-prone if it is not backed by suitable software engineering and structuring support [10] [11]. It is our belief that one should keep in mind several important issues related to system structuring while developing software diversity schemes.

First of all, software diversity should be introduced at the level of system structuring, so that the main structuring units (and, in particular, those of system design) should be the *units of diversity*. When applied generally, this approach allows us to use diversity recursively at several levels of system structuring [10]. This makes it possible to achieve a finer granularity of diversity and apply different levels of diversity in different units. If basic structuring units have general properties and there are features which make system design simpler, this automatically applies to system diversity. For example, many fault tolerance schemes separate specification and implementation, use inheritance, abstract data types, abstract/actual parameters.

Another critical issue is *diversity encapsulation*. It should be possible to apply a diversity scheme in such a way that components which use a diversely-designed (DD) component or contain it are ignorant of the fact that diversity has been employed. In addition, each version should not know about or refer to the rest of versions. Note that in the context of providing fault tolerance, encapsulation means clear error confinement and absence of erroneous information smuggling.

To successfully design complex systems with diversity, one should clearly *separate* different concerns in developing NVP schemes and in applying them. F, versions have to be designed and executed separately to enhance the independence of version failure modes. It is also important to separate the NVP control, adjudication and versions: they should have simple and clear interfaces and not exchange information at intermediate points of their execution.

Observing these general rules of *structured NVP* will simplify system design, reduce the complexity of systems and of using diversity, and give a lucid framework for applying system diversity in a disciplined way. This is how the principles of structured system design will operate in the NVP context This has not been much discussed as a separate issue and sometimes it is not realised that one should always follow them while developing software fault tolerance techniques. Some of these principles may look obvious but if the existing proposals for using NVP and its application are analysed it will be seen that not all of them take system structuring into account. For example, the concept of voting at intermediate points of version execution clearly contradicts these principles.

1.4. NVP and Object-Oriented Programming

Recently some research has been done [1] [11] [12] which has moved N-version programming into the object-oriented (OO) paradigm and demonstrated how it can be applied here. When this is done properly, diverse system development can benefit from OO programming features and can naturally follow structured NVP principles discussed above.

The main structuring units in OO development are classes (objects); this is why applying diversity at the level of classes is the most beneficial way [11]. Versions are to be developed to conform to the same class (type). For example, there can be several implementations of the list class, either static (a simple array or an array in which each element contains the index of the next element - the cursor implementation [13]) or dynamic (as one way ground, one way circular, two way ground and two way circular linearly linked structures [14]) ones, all conforming to the class above. A general framework for using software diversity in OO systems was proposed in [11]. In this framework, versions, adjudicators and the controller are classes. Diversity is hidden inside the diversely-designed class, and all versions have interfaces identical to its interface. This approach fits all principles of structured NVP well. The applicability of the general framework was demonstrated in the C++ language (although the authors had to resort to an unspecified underlying mechanism to allow concurrent execution of versions). This implementation is presented as a set of re-usable classes.

The first approach [12] to introduce class diversity was developed for the Arche language. This scheme does not hide diversity, so the user has to program many functions of the controller; in particular, s/he explicitly declares version objects and calls N version objects and the adjudicator. The approach relies on an experimental run-time support which takes care of version concurrency and synchronisation; among other things, this support has a special feature for concurrent call of several methods.

Our class diversity scheme [1] relies on the general framework developed previously at Newcastle University [11]. The intention is to develop a NVP scheme suitable for concurrent OO languages: we want to explicitly address all difficult questions of version synchronisation and concurrency and to develop re-usable components which perform these functions. Another purpose is to propose a practical scheme which would be implementable in many widely-used concurrent OO languages and demonstrate this using Ada.

We have been using this scheme as a testbed for investigating different topics related to NVP in OO systems. One of these is a clear separation of the responsibilities of the system programmer, the fault tolerance (FT) programmer and version programmers; in particular, we discuss the order in which different components are to be developed, at which phases and how these programmers cooperate. Another issue is understanding the re-usability of general NVP components: unfortunately, not all components in [11] can be easily re-used in practice (for example, our analysis [1] shows that it is hardly possible to re-use adjudicators in OO languages because they are always output-parameter-specific).

Object-orientation offers many advantages in developing and using software diversity: first of all, decreasing the cost and improving the ability to deal with complex system design by re-using application and control software and by imposing structured design. Another advantage of the proposals in [1] [11] is their orientation towards standard OO languages (paper [1] takes this further because the controlling mechanism is developed at the language level and made re-usable).

In the next section we will introduce the system exception handling model we will be using throughout the paper. Section 3 will discuss our motivations for introducing exception handling into NVP and outline the framework proposed. In Section 4 we will briefly describe our Ada implementation of the framework. The next section will analyse our scheme an compare it with the existing NVP approaches. The conclusions and the outline of our future work will be presented in Section 6.

2. Exception Handling Model

In our exception handling model, each object (class) can have several *interface (external) exceptions*: E^1, E^2, E^3, ..., which can be propagated from it: this happens when an exception is signalled inside the object while one of its methods is being executed. Object implementation can have several *internal exceptions* (e^1, e^2, e^3, ...): these are declared, raised and have to be dealt with inside. The object is the exception context which has to have handlers for all internal exceptions. Although there are programming languages which allow handlers to be attached only to blocks of statements (Java, C++, Arche), in many of them (e.g. Ada, Eiffel) handlers can be attached to the method bodies. We have chosen a more general OO model which allows associating handlers with an object as a whole as well (the Lore [15] and extended Ada [16] languages follow this model). Paper [15] argues the importance of this feature and discusses the flexibility the designers gain by using it, [16] discusses practical evidence which backs this approach. Note that our model can be easily simulated in all of the above mentioned OO languages.

In our model, internal exceptions and their handling are encapsulated inside the object. Its execution can be completed either normally (which means that external exceptions have not been signalled, although an internal exception might have been raised and successfully handled) or exceptionally, when an external exception has been signalled (either from the normal code or from an internal exception handler). A pre-defined Failure interface exception is used when the object is not able to provide any consistent results and signal any interface exception. We follow the termination model [3] here: the execution of the context does not continue after an exception has been handled and the execution of the object is completed after any handler has been executed. We believe that it is vitally important to separate internal and external exceptions and not to allow internal exception to be propagated from the object. We impose the following restrictions on the object design: all internal exceptions have to have handlers, all external exceptions have to be specified in the class specification because they are part of the class interface/behaviour. These rules of safe exception handling can be easily checked in compiler time.

External exceptions are signalled to the containing exception context to inform it of an object being not able to produce the result required. These exceptions have to be internal for each object that uses this object. Signalling external exceptions does not necessarily mean that there has been a failure in the execution of the object. We take here a wider view on exceptions, assuming that the object is in an abnormal state if it has not been not able to produce the required result in its entirety. There can be many interpretations of the interface exceptions and many ways of using them. In particular, they can:

• report that the object has not been changed (the abort exception);

• signal partial results delivered by the object (we may need parameterized external exceptions in this case but we do not include them in our framework at the moment);

• inform the containing context of a fault in the environment (including other objects, resources, underlying services, etc.) which the object has detected, but is not able to handle;

• report that the method call is incorrect (e.g. input parameters) - interface exceptions;

• inform the containing context that object pre-conditions, invariants or post-conditions have been violated;

• signal that a design fault has been triggered in the object;

• inform the containing context of the object having been left in an intermediate state because of an error or because it has not been not possible to undo all changes.

Predefined exceptions of an object are the external exceptions of the objects it uses. They cannot be explicitly raised by the object, but an attempt has to be made to handle them (all of them have to have handlers).

Four scenarios are possible while an object is being executed (Figure 2):

• no exceptions are raised or signalled; the object outputs normal outcome;

• an internal exception e^1 is raised and successfully handled inside the object, which reports normal outcome;

• an internal exception e^3 is raised and the corresponding handler tries to handle it but fails, so an external exception E^2 is signalled to the containing context;

• an external exception E^1 is signalled by the component without raising any internal exception.

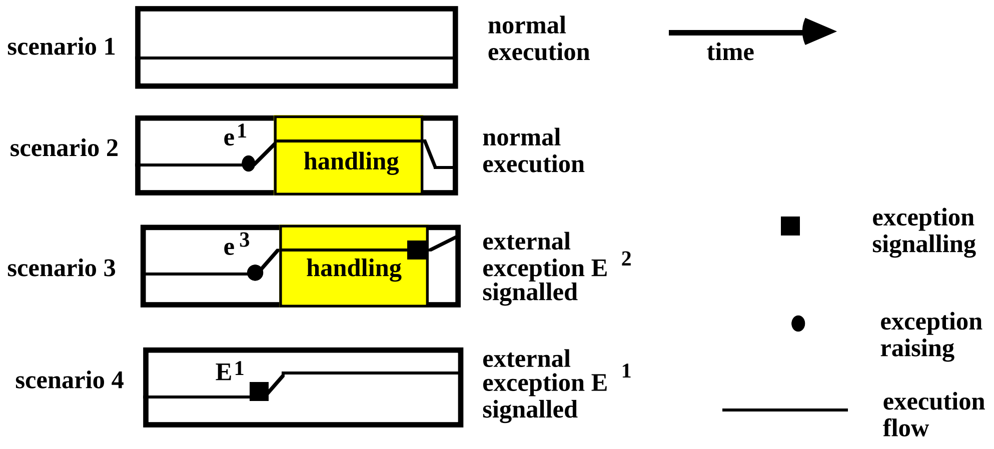

Figure 2. Execution of a component

3. NVP and Exception Handling

3.1. Motivations. Intentions

Exception handling is a general structured approach for dealing with abnormal situations which has been extensively researched and widely used in many industrial applications. We believe it is vital to make it possible for exception handling to be used together with N-version programming.

First of all, this would allow DD components to be employed in a general context in which exception handling is used for developing components without diversity. That means that DD components should be able to have exceptions in their interfaces in the way conventional components do. Secondly, version designers should be able to use exception handling while developing versions and to deal with exceptions signalled by components they use in their turn. It is our belief that it is crucial to utilise all appropriate means available while fighting bugs and providing fault tolerance. Every measure should be used inside each version: we have to try and tolerate as many faults as possible at this level. The most general way for achieving fault tolerance is *forward error recovery* [8] which is usually implemented using exception handling features. If we can use exception handling together with NVP, we have more chance of continuing execution and delivering results.

Let us consider a simple example. It would be normal practice to use external exceptions in the class list shown above; exception list_empty would be used to inform the caller that there is no element in the list when, for example, method Get_Elem is called. It is clear that in NVP all versions should report this exception if they function properly and that the adjudicator should choose this external exception as the correct result to be reported outside. We need a new approach to allow adjudication in all situations (including the situation when versions cannot deliver normal results).

Unfortunately, NVP and exception handling are usually discussed separately, and there is no general object-oriented framework developed for extending NVP by exception handling. Our intention is to put this right.

There are only two papers which address this topic [11] [12]. To some extent, our research is based on paper [12], which presents the first scheme combining NVP and exception handling. We believe that this is a very important scheme because it is object-oriented, so that diversity is introduced at the level of classes and because it allows exceptions, signalled from separate versions, to be combined in one resolving exception which is signalled further. Unfortunately, this scheme is not general, it is oriented towards a particular language and a particular executive. We have found many serious problems with it; for example, the exceptional results of all versions are not voted but, rather, all taken into account (which clearly contradicts the idea of NVP), another problem is that of correct system structuring: diversity is not hidden in this scheme and the caller has to call versions and deal with their (exceptional and normal) results explicitly (see Section 6 for a detailed discussion of this scheme and comparison).

Paper [11] proposes a general framework for introducing software diversity into OO programming. A system can be built out of idealised fault tolerant components with diverse design: one can, in particular, use NVP in this framework. Each version can have a failure exception to be signalled outside and handle internal exceptions. A component can signal a failure exception if there is no majority agreement. It is a very general framework which does not consider how failure exceptions of versions are to be dealt with. We believe that there are some problems with structuring software in this framework. It does not seem to be right to allow handlers for internal exceptions to be seen from the outside and to have handlers for the same exceptions in all versions.

Another problem is that there is no clear separation between internal and external exceptions. External exceptions are not part of class specification in this framework. It is clearly not enough to be able to signal and deal with only failure exceptions of versions and of the whole component. The idea of combining exception handling and NVP needs to be developed further than in this proposal.

Our intention is to propose a general exception handling framework for NVP (some initial ideas were reported in [17]) which would allow version designers to use internal exceptions in developing versions, versions to reach a majority agreement when several of them are not able to produce normal results and the system to continue execution after some of the versions have signalled exceptions. This scheme should allow a clear separation between internal and external object exceptions, and overcome many of the problems in the existing schemes. This will allow execution to continue and the DD class to be used even after an exception has been signalled. To do this, we have to guarantee that the states of the majority of versions are the same.

We need to develop an approach which would combine: structured system design, exception handling and structured NVP in a natural and simple way. This would simplify the design of systems with diversity; otherwise, the complexity of systems will grow as well as the difficulty of using diversity. This framework will be developed as an extension of our original NVP scheme [1].

3.2. General Description of the Framework

The framework will answer the following questions:

• what are internal and external exceptions for NVP?

• how can DD components have exceptions in their interface in the same way as components without diversity do? How can DD components be included into the general system structure in which external exceptions are used?

• how can version designers use exception handling to improve version design? How can they use both internal and external exceptions?

• how are normal and exceptional version outcomes adjudicated?

In our approach each version has both internal and external exceptions. Internal ones are specific for the version, external ones are the same for all versions (they are defined by the class interface) and for the DD component as a whole. We follow the termination model here. If the handling of an internal exception inside a version has been successful, the version is completed normally as if nothing happened (transparently for the rest of the versions and for the NVP controller), if not, a version interface exception is signalled to the NVP controller. The execution of each separate version always follows one of the four patterns presented in Figure 2. After the version is completed, it can be either in a normal state or in an exceptional state (corresponding to the interface exception signalled). We do not interrupt the remaining versions even if a version signals an exception because normal correct results can be produced even after this and because we need all results to be able to tell which ones are in the majority, i.e. correct.

In our scheme each version can decide to signal any interface exception. But it is clear that it would be wrong to propagate it directly outside: we need adjudication here. After all versions have completed, the NVP controller calls the adjudicator which should have extended functionalities to take a majority decision in all possible scenarios (including the exceptional ones). It decides which versions have produced erroneous results (which means that these versions are considered to be faulty because they have software bugs). A version is either in a normal or in an exception state after its execution has been completed. After the adjudication we can categorise each of them as being either correct or faulty.

Adjudication of exceptions requires a consensus on exceptional outcomes. We can have a consensus when more than half of the versions have signalled the same exception. If the majority delivers the same normal result, the adjudicator reports it; if the majority signals the same exception, it reports it, and the controller signals it. If a version is in the minority, its results are considered erroneous and its state faulty. If there is no majority, we signal a pre-defined failure exception. In this case we should ignore this DD component in the future unless a special repair procedure has been performed to correct the states of all versions and put them into a consistent state. Note here that, generally speaking, this repair is impossible to provide from the containing context because diversity is hidden inside the DD object (except, in some cases, for re-initialisation).

The approach allows using the abort interface exception, which is viewed as one of the interface exceptions: if the majority of versions guarantee the "nothing" semantics and signal the abort exception, then it will be signalled to the containing context.

3.3. Diversely-designed Components

We propose to develop DD classes in the way shown in Figure 3. The functionality and interfaces of all components are extended to accommodate exception handling (compare with the traditional model shown in Figure 1).

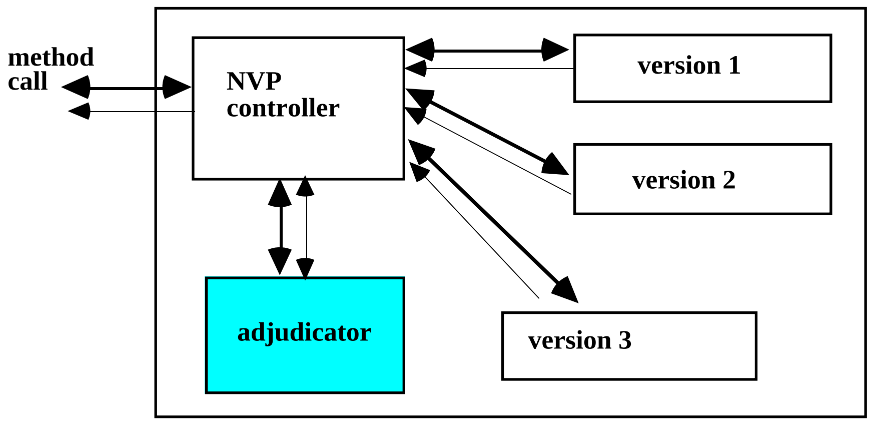

Figure 3. The architecture of the diversely-designed object (the fine lines show the flows of information related to exceptions)

The DD component looks like a normal component from the outside (including the fact that it has interface exceptions): diversity is hidden. Its implementation consists of the controller, adjudicator and versions. The controller, having the interface of the DD component, gets control each time a method of the DD object is called and calls versions and the adjudicator when necessary. The controller has the usual functionalities [1] related to version and adjudicator control (including the forking of N threads and joining them after all versions are completed). Apart from these, it has special functionalities: collecting exceptions signalled by versions, passing them to the adjudicator, signalling the adjudicated interface exception (if necessary), and is application-independent. The adjudicator is called by the controller after all versions have completed their execution. The DD object interface and version interfaces are the same, so that they can propagate only the same external exceptions.

3.4. Internal Exceptions of Versions

Each version should be developed independently (we view it as a usual component), so its execution follows one of the four patterns described in Section 2. The development of versions does not differ from that of normal components (generally speaking, their developers should know nothing about diversity).

In our approach each version V^i can have internal exceptions: $e^{i,1}$, $e^{i,2}$, ..., e^{i,k_i}, and corresponding handlers $eh^{i,1}$, $eh^{i,2}$, ..., eh^{i,k_i}, declared inside V^i. These exceptions cannot be propagated outside. The version programmer uses two operations: **raise** $e^{i,j}$ to raise internal exception $e^{i,j}$, and **signal** E^l to propagate external exception E^l from the version. We follow the termination model here: handler $eh^{i,j}$ either succeeds in handling exception $e^{i,j}$, in which case the version is completed and reports the normal outcome (by returning the control and the output parameters, if any), or the version signals an interface exception from handler $eh^{i,j}$ which propagates to the NVP controller (this completes version execution as well).

Internal exceptions are specific for each version, they and their handlers are to be developed by independent version programmers. There is no need and no way to coordinate the internal execution of versions because we do not impose any restrictions on version design. Versions are ordinary components whose developers are not aware of the fact that they are versions of a DD component. But these components are glued together in the usual way.

After a method has been executed, a version can be either in a normal or in an exceptional state (after an exception has been signalled outside) corresponding to the exception signalled.

3.5. External Exceptions of Versions

The interfaces of the DD object and of versions contain the same exceptions : E^1, E^2, E^3, ..., E^m, Failure (exceptions list_empty, list_full, list_absent_element, list_failure of the list class above). Any version can either produce a normal outcome or signal an interface exception. The NVP controller waits for the completion of all versions. To improve error detection, we recommend using time-outs: external exception Failure is assumed to be propagated by a version when it breaks the corresponding time-out. This is only one of the situations when version exception Failure is used in our approach. It is meant to signal any serious problem found during version execution: e.g. the supporting mechanism does not work properly, it is not possible to ensure any post-conditions or consistency in the version state, etc. Exception Failure is treated by the controller and by the adjudicator in a special way. If the version state is not repaired or recovered, it cannot be used any more and its results cannot be trusted.

3.6. Exception Adjudication

External exceptions of versions cannot be directly propagated into the context of the caller: they have to be adjudicated. After all versions have been completed the controller passes their results to the adjudicator. This is an extended adjudicator which can deal with exceptional outcomes signalled by versions; its purpose is the same as that of the conventional one: to find the majority result. In addition, the adjudicator performs error detection by finding the versions which produced that, as well as the versions which produced minority results and are assumed to be faulty because they have had faults triggered but not detected and tolerated inside (they propagated erroneous results). We distinguish between exceptional and erroneous outcomes here. We treat version results as erroneous if this version has been found to be in the minority. Exceptional outcomes do not necessarily mean errors, and they are not in any way related to the errors which NVP is intended to mask: the exceptional outcome found to be in the majority is simply propagated to the containing context and dealt with at another level (one of the possible scenarios is shown in Figure 4). Versions which reported them are not in erroneous states, apart from the situation when the majority of versions have signalled the Failure exceptions.

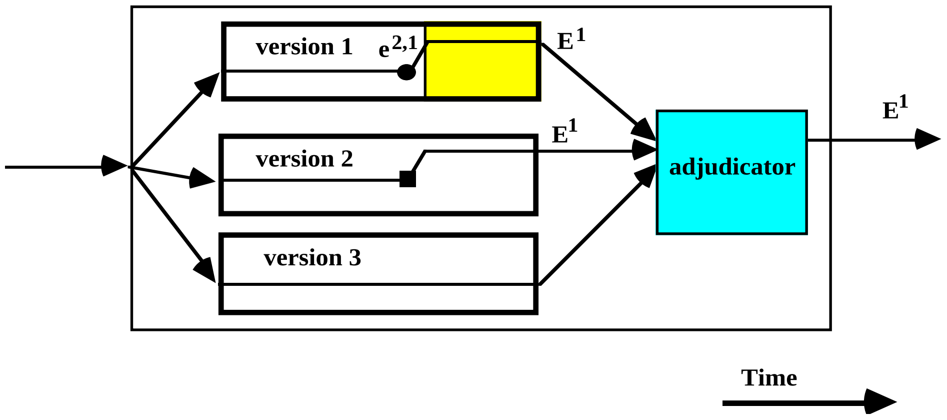

Figure 4: Adjudication of exceptions

Pre-defined Failure exception is reported by the adjudicator when there is no consensus on outcomes. Any version can signal Failure exception, in which case the adjudicator operates in the usual way: if a majority of versions have signalled the Failure then the adjudicated result is the Failure exception, otherwise the result of the failed version is ignored (masked). Another situation when the adjudicator reports the Failure, exception to the controller is when it is not able to perform the adjudication properly (for example, because of a design fault).

Any conventional adjudicator [18] can be used for dealing with normal version outcomes but it has to be extended for our scheme by *exception adjudication.* Note that this adjudication is simple because there is a finite number of interface exceptions in any DD class and their comparison is straightforward when they have unique values associated with them; this is why the exact majority voting [18] can be applied.

Let us consider several examples demonstrating how the extended adjudicator works for the system with three versions:

• outcomes (E^1, E^1, Normal) -> result E^1; (outcome Normal is produced by a faulty version, exceptional outcome E1 is produced by the correct versions);

• (E^1, E^2, E^1) ->E^1;

• (E^1, E^2, E^3) ->Failure;

• (E^1, E^2, $Normal^2$) -> Failure;

• ($Normal^1$, $Normal^2$, E^1) -> Failure;

• (Failure, Failure, Normal) -> Failure.

3.7. External Exceptions of the DD Class: Exception Propagation

The controller signals the adjudicated exception to the containing exception context (e.g. to the caller component) which has to deal with this exception. To do this in our model, the caller has to have handlers for all exceptions which can be signalled by the DD component (actually, by all components it calls).

Note that, in our model, exception Failure of the DD component is conceptually identical to exceptions Failure of versions: it is signalled when it is not possible to provide any trustworthy results and when the component is left in an inconsistent state (at the level of the DD component this means that either there is no majority or a majority of versions have signalled Failure exceptions).

We can summarise the scheme by showing transitions of version states (see Figure 5). During version execution it can be in either "normal inside" or "handling" states. After the execution is completed, the version can be in either "normal outside" or "exceptional" state. The subsequent adjudication tells whether this state is correct or faulty.

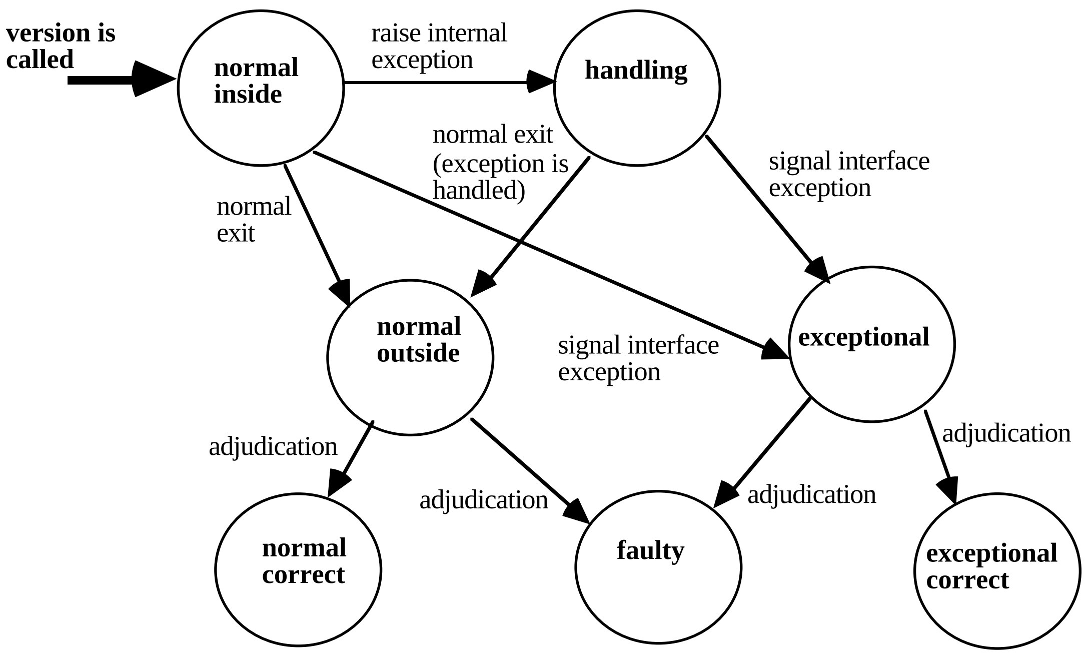

Figure 5. State transitions of a version

4. An Ada Implementation Scheme

We use Ada [4] for demonstrating the approach proposed for a lot of reasons: thus, this is the first standard OO language with the concurrency feature (which allows us to program the NVP control explicitly and make it re-usable); this language is used in many critical areas in which software diversity is employed. Rather than designing an exotic system and language with constructs for NVP, we have chosen to demonstrate our approach through an existing language which is widely used in industry as well as in education. We believe that, thanks to this choice, our scheme can be tried in practical systems straight away.

This implementation is based on the Ada NVP scheme [1] which implements the traditional NVP without exceptions. The original scheme has been considerably modified to allow each version to signal exceptions, to accommodate a new adjudicator and new functionalities of the controller. In addition, we have changed the information flows between the components of the NVP scheme (Figure 3), made the scheme more reusable by introducing new abstract classes and facilitated the FT programmers' job by simplifying the patterns they have to follow in programming the NVP controller.

The abstract list class (Section 1) is used for deriving the DD class (instances of which are declared and used by the caller):

```
package ft_list_class is
       type ft_list_t is tagged limited private;

       procedure Initialize (l : access ft_list_t);
       procedure Insert (l : access ft_list_t; elem : in elem_t);
       procedure Remove (l : access ft_list_t; elem : in elem_t);
       procedure Get_Min (l : access ft_list_t; elem : out elem_t);
       procedure Sort (l : access ft_list_t);
       function Check (l : access ft_list_t; elem : in elem_t) return boolean;
```

```
private
       type ft_list_t is new list_t  with
              record
                nvp_c: NVP_controller(version_max); -- controller
              end record;
end ft_list_class ;
```

and for deriving version classes. The DD class implements the NVP controller, and has N version objects and the adjudicator object declared inside. N tasks are forked by the controller when a method of the DD object is called. Each task has a similar structure:

```
task body Get_Min_LV1 is
       My_number: constant :=1;     -- number of this version
       elem_out: elem_t :=default;
       -- ...
begin
       begin
         Get_Min (List_V1'Access, elem_out); -- call of the version method

         -- pass the results to the adjudicator:
         l_adj.Get_Min_Keep_Outs(My_number,Ada.Exceptions.Null_Id, elem_out);

         exception
              when the_occurence : others =>
                -- pass the exception identity to the adjudicator:
                l_adj.Get_Min_Keep_Outs(My_number,
                            Ada.Exceptions.Exception_Identity(the_occurence));
       end;
       -- version state recovery using an abstract state of the correct version
       -- if the version found to be faulty
end Get_Min_LV1;
```

Exceptions have unique identities in Ada: the controller catches all exceptions signalled by versions and passes their identities to the adjudicator. The identity of the majority exception is returned to the controller to be signalled outside (using the conventional Ada **raise** statement). Internal version exceptions are used independently by the designer of each version following Ada rules with some additional restrictions made necessary by the fact that the Ada exception model does not exactly fit to our model because its space of exceptions is flat. This model, for example, allows transparent exception propagation from the methods and does not require all internal exceptions to have internal handlers.

Our implementation is delivered as a set of re-usable components (classes, protected objects, etc.) and templates to be followed by version and FT programmers.

5. Discussion

5.1. Specification and Use of External Exceptions

In our model each version programmer decides which internal exceptions to use and how to handle them. The situation is quite different with respect to external exceptions of a DD object because these are used by different version designers.

Let us consider an example. If we do not rigorously specify conditions when exceptions for the list class above are signalled then two external exceptions list_empty and list_absent_element, can be signalled by two correct implementations of the Remove method in response to the same call if the list is empty. Clearly this should be avoided because in this and similar situations the adjudicator is not able to find a consensus interface exception; obviously, we would like the NVP scheme proposed to be able to deal with them.

This example demonstrates the importance of non-ambiguous specification of external exceptions. The specification of each external exception should include a rigorous description of conditions (including inputs) in which it is to be signalled (the *exceptional conditions*) and a description of the state the object is left in and of the outputs produced, if any (the *post-conditions*). In the example above the exceptional condition should state that exception list_absent_element is signalled by method Remove when two conditions are met together: there is no such element in the list and the list is not empty.

Exceptional conditions for external exceptions should be orthogonal (non-overlapping) and cover all situations of which we would like to inform the caller after the execution of a DD object. Only this can guarantee that the same exception is always signalled by all correct versions and that there is an exception for each abnormal situation of which the DD object might wish to inform the caller. An exception can be signalled by a version only after it is guaranteed that its internal state satisfies the corresponding post-conditions. Note that a failure by some versions to identify correctly exceptional conditions is treated as a design fault in our model and will be automatically adjudicated within the scheme proposed.

We believe that these requirements are of general importance for any set of external exceptions associated with the class. System design and analysis is essentially facilitated when exceptional and post conditions are rigorously defined for each exception of the set, when these conditions are orthogonal and together cover all possible exceptional states. To deal with external exceptions properly, the caller definitely has to have a complete (albeit abstract) description of the state the object has been left in.

Another important point is that the states of all correct versions should be the same if they signal the same exception. This means that their states satisfy the same post-conditions. For example, they may all use the "nothing" semantics for some exceptions, in which case the post-conditions are equal to the pre-conditions before the method call. This rule is important because the caller treats the DD object as a whole and it may respond to an external exception by performing some operations on this object (for example, to compensate for the exception or to recover the object), in which case all versions should respond to these operations in the same way.

Exceptional conditions, post-conditions and conditions ensuring that the states of all correct versions are identical should be expressed in general terms, using an abstract view of the DD object without any references to implementation details. Version programmers have to ensure these conditions. At this stage we suggest using English for describing conditions but in the future this will be done in a formal way, with automatic checks of the required properties of specifications.

5.2. Comparison

In this section we compare our proposal with the schemes discussed in [12]. The OO language Arche [12] first introduced exception handling into NVP. We have used some ideas from this scheme but found many problems which we have tried to overcome in our scheme. First of all, this scheme heavily relies on a special run time support which provides a concurrent method call feature. Secondly, it does not hide diversity inside the DD class, which means that the caller has to declare all version objects and the voter, and to deal with the outcomes of all versions.

Our analysis shows that this scheme contradicts the idea of NVP and some of the principles of structured NVP discussed above. Firstly, in this scheme the execution of versions is interrupted should any of them signal an exception. We believe that the approach taken in Arche conceptually disagrees with the idea of majority voting and that it complicates the use of diversity tremendously because one cannot guarantee any consistency in the states of version objects whose execution has been interrupted: it is not at all clear how programmers can deal with such versions in subsequent computation. One more contradictory idea behind this scheme is that of resolving all exceptions signalled by all versions: if we are to follow the intention of NVP we should vote here and ignore the minority results (including exceptions) as produced by faulty versions. Another reason why we believe that using exception resolution [19] is not adequate here is that when all exceptions signalled by versions are resolved and a covering exception (a "concerted" one in the terminology adopted in [12]) is calculated, the states of versions, generally speaking, do not correspond to this exception and are inconsistent. This is because the state of a version corresponds to the exception it has signalled and not to any other exception (e.g. to a "resolving" one).

Another of our concerns is that the Arche approach insists on recovering the internal version state of some versions from the outside by the user of the DD object. This is to be done by special exception handlers at the level of the caller: these handlers should access the state of these versions via standard interfaces. We believe that one cannot do this properly in this way because it must be assumed that the whole state of the faulty version (the one found to be in the minority) is corrupted and that we cannot trust the results it has produced (including, for that matter, the exception signalled which in [12] is used to choose the recovery measure). The only correct way of recovering a version is to use an abstract intermediate representation of the state of the correct version [20].

Our analysis shows that exception handling and faulty version recovery features are intended for different purposes in NVP and that, generally speaking, they cannot replace each other. If after the adjudication we find that a version is faulty we should assume that its entire state is corrupted, in which case the only realistic recovery is by mapping the state of a correct version onto the state of the faulty one. This recovery cannot be done at the caller's level if we want to hide the diversity inside the DD class. We will not be able to handle an external exception signalled by a majority of versions using faulty version recovery features which are intended for recovering versions found to be in the minority. This is because an external exception signalled by the DD object and the fact that some versions are in the minority (and, as a result, faulty), which are separate events in our model, should be treated differently and at different levels.

6. Conclusions

This work has been built on the research done in Newcastle on developing a general exception model for complex concurrent systems structured using Coordinated Atomic actions [21].

NVP is one of the most suitable techniques for implementing real time fault tolerance [22]. The approach proposed will facilitate fault tolerance design of real time OO systems for the following reasons. First, using internal version exceptions not only improves the ability to tolerate faults but makes recovery faster because it is application-specific and is performed at the level of separate versions. Second, using interface exceptions in the DD class gives better diagnostics for any object which uses an object of this class and by this makes their handling faster and simpler.

In our future research we plan to adopt one of the formal approaches (e.g., [2] [23]) allowing formal reasoning about systems with exceptions, to specify exceptional and post-conditions of versions signalling exceptions. This should make it possible to automatically check the required properties of sets of exceptions and of corresponding conditions, e.g., to analyse their completeness, correctness and disjointness (Section 5.1).

Another direction of the future research is implementing a distributed Ada scheme (note that Ada is the first standard OO language which has features for distributed programming [4]). We will start with establishing a new set of failure assumptions (which will have to include some hardware faults, e.g. node crashes). Some of the problems which we will have to tackle are: breaking diversity encapsulation, declaring version objects on different nodes ("partitions" in the Ada terminology) and controlling them distributedly, using time-outs to watchdog all distributed accesses, finding a way to pass exception values between partitions (exception identities can be used only locally in Ada).

In this paper we have developed an extended NVP approach enriched by exception handling. It is based on the principles of structured NVP, which we have formulated, and provides two important features: diversely-designed classes can have external exceptions in their interfaces, versions can use exception handling internally (including dealing with exceptions propagated from the objects they use). The approach is generally applicable and can be implemented in any OO language or system which has exception handling features. We have demonstrated this with an Ada implementation developed as a set of re-usable components and templates for the programmers to follow.

Acknowledgements. This research has benefited tremendously from the join research on exception handling done with my colleagues: B. Randell, J. Xu, A. Zorzo. It has been supported by the EPSRC/UK DISCS ("Diversity in Safety Critical Software") project.

References

1. Romanovsky, A. (1999) Class Diversity Support in Object-Oriented Languages. *Journal of Systems and Software*, **47**, 43-57.

2. Cristian, F. (1994) Exception Handling and Tolerance of Software Faults. In Lyu, M. (ed), *Software Fault Tolerance.* Wiley.

3. Goodenough, J. (1975) Exception Handling, Issues and a Proposed Notation. *CACM*, **18**, 683-696.

4. Intermetrics (1995) Information technology - Programming languages - Ada. Language and Standard Libraries. ISO/IEC 8652:1995(E). Intermetrics, Inc.

5. Liskov, B. H. and Snyder, A. (1997) Exception Handling in CLU. *IEEE TSE,* **SE-5**, 203-210.

6. Randell, B. (1975) System Structure for Software Fault Tolerance. *IEEE TSE,* **SE-1**, 220-232.

7. Avizienis, A. (1985) The N-version Approach to Fault Tolerant Systems. *IEEE TSE,* **SE-11**, 1491-1501.

8. Lee, P. A. and Anderson, T. (1990) *Fault Tolerance: Principles and Practice.* Springer-Verlag, Wien - New York.

9. Lyu, M. and Avizienis, A. (1992) Assuring design diversity in N-version software: a design paradigm for N-version programming. In *Proc. of the 2nd Dependable Computing for Critical Applications*. 197-218.

10. Randell, B. (1983) Recursive Structured Distributed Computing Systems. In *Proc. of the Third Symposium on Reliability in Distributed Software and Database Systems.* Florida, USA, 3-11.

11. Xu, J., Randell, B., Rubira, C. M. F. and Stroud, R. J. (1995) Toward and Object-Oriented Approach to Software Fault Tolerance. In Avreski, D. (ed), *Fault-Tolerant Parallel and Distributed Systems.* IEEE CS Press,

12. Issarny, V. (1993) An Exception Handling Mechanism for Parallel Object-Oriented Programming: Towards Reusable, Robust Distributed Software. *J. of Object-Oriented Programming,* **6**, 29-40.

13. Weiss, M. A. (1992) *Data Structures and Algorithm Analysis.* The Benjamin PC, Inc.,

14. Beidler, J. (1997) *Data Structures and Algorithms.* Springer,

15. Dony, C. (1990) Exception Handling and Object-oriented Programming: Towards a Synthesis. In *Proc. of the ECOOP/OOPSLA'90*. 322-330.

16. Cui, Q. and Gannon, J. (1992) Data-oriented Exception Handling. *IEEE TSE,* **SE-18**, 393-401.

17. Romanovsky, A. (1999) On N-Version Programming and Exception Handling. In *Proc. of the 10th European Workshop on Dependable Computing (EWDC-10)*. Vienna, Austria, 175-179.

18. Di Giandomenico, F. and Strigini, L. (1990) Adjudicators for Diverse Redundant Components. In *Proc. of the 9th Int. Symp. Reliable Distributed Systems*. Huntsville, Alabama, 114-123.

19. Campbell, R. H. and Randell, B. (1986) Error Recovery in Asynchronous Systems. *IEEE TSE,* **SE-12**, 811-826.

20. Tso, K. S. and Avizienis, A. (1987) Community Error Recovery in N-Version Software: a Design Study with Experimentations. In *Proc. of the FTCS-17*. Pittsburg, USA, 127-133.

21. Xu, J., Romanovsky, A. and Randell, B. (1998) Exception Handling and Resolution in Distributed Object-Oriented Systems. In *Proc. of the ICDCS-18*. Amsterdam, The Netherlands, 12-21.

22. Kim (Kane), K. H. (1995) Major Research Issues in Real-Time Fault-Tolerant Computing. In *Proc. of the Pacific Rim Int. Symposium on Fault-Tolerance*. Newport Beach, California, USA, 141-143.

23. Bidoit, M. (1984) Algebraic Specification of Exception Handling and Error Recovery by Means of Declarations and Equations. In *Proc. of the 11th Colloquium on Automata, Languages and Programming*. Antwerp, Belgium (LNCS-172), 95-108.